.NET Full Stack Development Program

Day15 ASP.NET Razor

Outline

- Razor
    - Razor Syntax
    - Razor Directive
    - Flow Control in Razor
    - Comment in Razor
- Razor Pages
- Razor Pages vs. MVC in ASP.NET

Razor

What is Razor

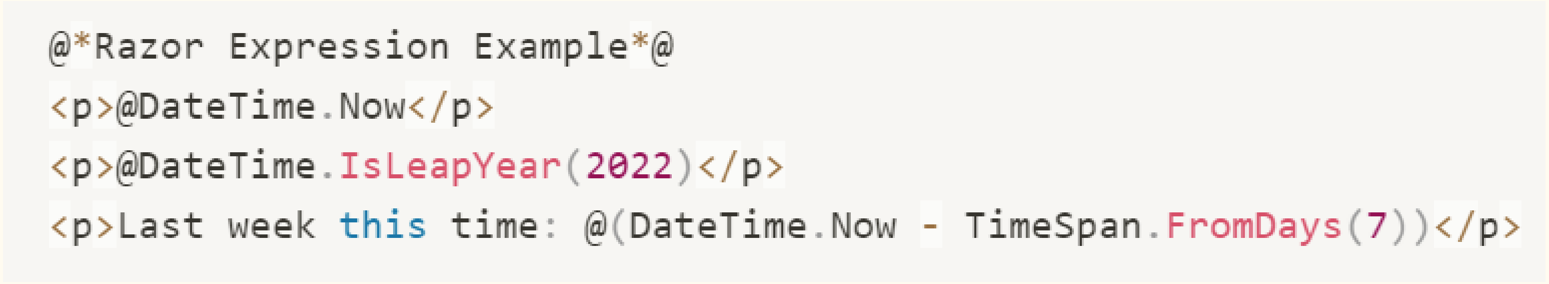
- Razor is a markup syntax that lets you embed server-based code (Visual Basic and C#) into web pages.

- Server-based code can create dynamic web content on the fly, while a web page is written to the browser. When a web page is called, the server executes the server-based code inside the page before it returns the page to the browser. By running on the server, the code can perform complex tasks, like accessing databases.

- Razor is based on ASP.NET, and designed for creating web applications.

Razor Syntax

- The Razor syntax consists of Razor markup, C# and HTML.
- Files containing Razor generally have a .cshtml file extension.
- Razor supports C# and uses the @ symbol to transition from HTML to C#. Razor evaluates C# expressions and render them in the HTML output.
- Razor epressions start with @ followed by C# code

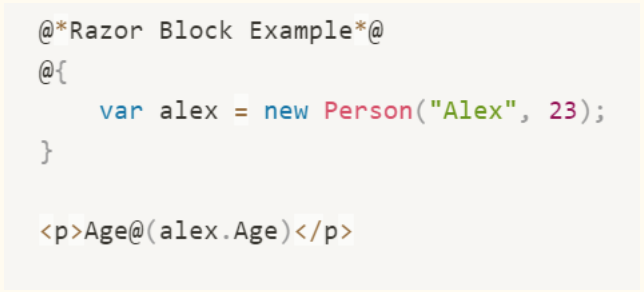


```
@*Razor Expression Example*@
<p>@DateTime.Now</p>
<p>@DateTime.IsLeapYear(2022)</p>
<p>Last week this time: @(DateTime.Now - TimeSpan.FromDays(7))</p>
```

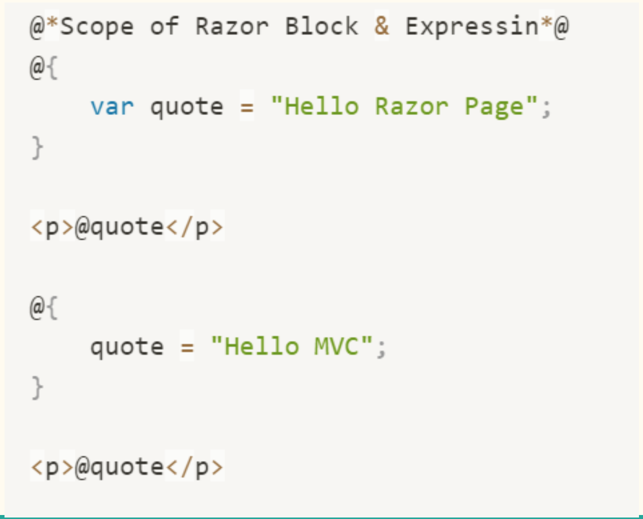
Razor Syntax

Razor Blocks

- Razor blocks start with @ and are enclosed by {}
- Unlike Razor expressions, C# code inside the block isn’t rendered.
- Code blocks and expressions in a view share the same scope and are defined in order.



```
@*Razor Block Example*@
@{
    var alex = new Person("Alex", 23);
}

<p>Age@(alex.Age)</p>
```

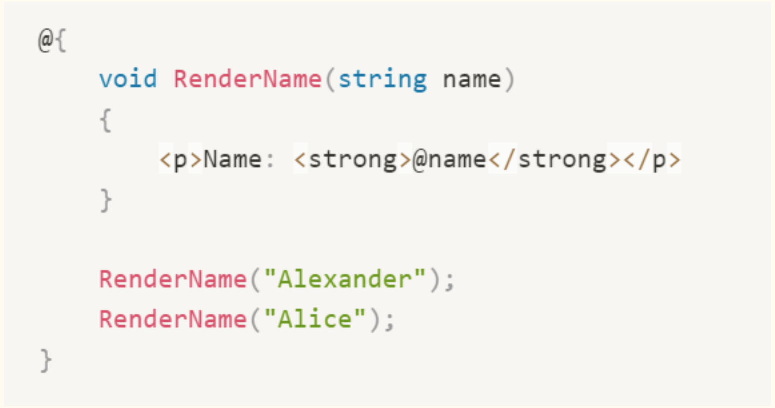


```
@*Scope of Razor Block & Expressin*@
@{
    var quote = "Hello Razor Page";
}

<p>@quote</p>

@{
    quote = "Hello MVC";
}

<p>@quote</p>
```

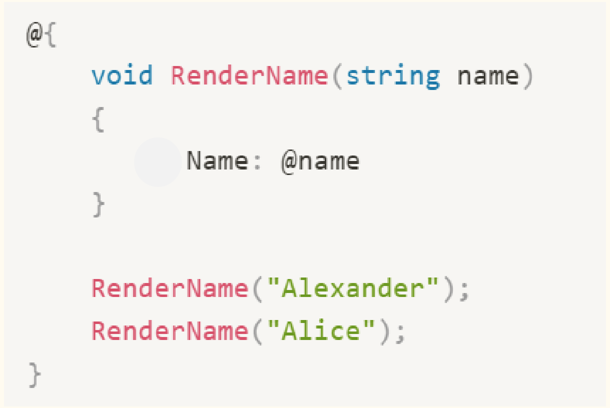
Razor Syntax

Razor Blocks

- The default language in a code block is C#, but the Razor Page can transition back to HTML.
- To render the HTML in a block we can use tags, only the content between <tag> is rendered; The @: syntax also works



```
@{
    void RenderName(string name)
    {
        <p>Name: <strong>@name</strong></p>
    }

    RenderName("Alexander");
    RenderName("Alice");
}
```



```
@{
    void RenderName(string name)
    {
        Name: @name
    }

    RenderName("Alexander");
    RenderName("Alice");
}
```

Razor Directives

Razor directives are represented by implicit expressions with reserved keywords following the @ symbol. A directive typically changes the way a view is parsed or enables different functionality.

- @page

The @page directive in a .cshtml file indicates that the file is a Razor page

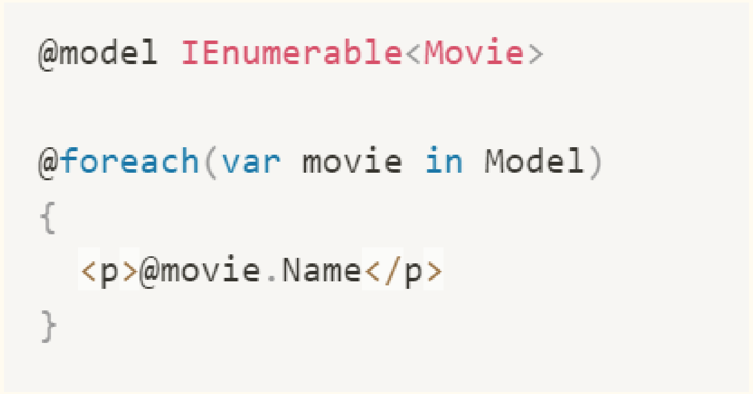
- @using

The @using directive adds the C# using directive to the generated view
i.e. if you put @using System.IO in your razor page, then you can use all the methods within System.IO namespace at this page.

Razor Directives

- @model

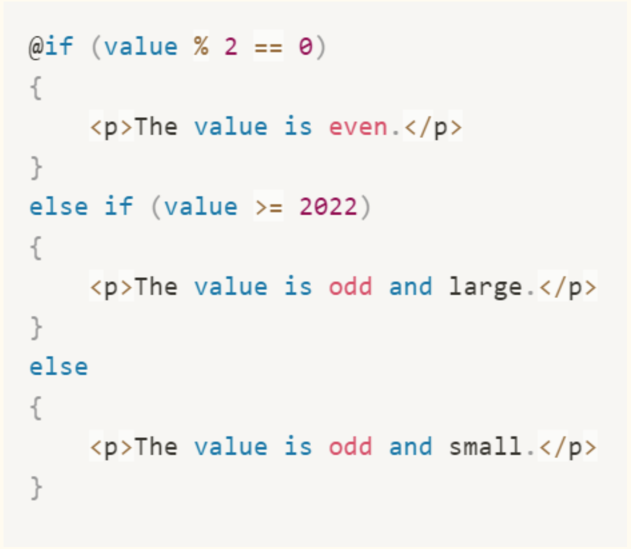
The @model directive specifies the type of model passed to a view or page, and Razor exposes a **Model property** for accessing the model passed to the view



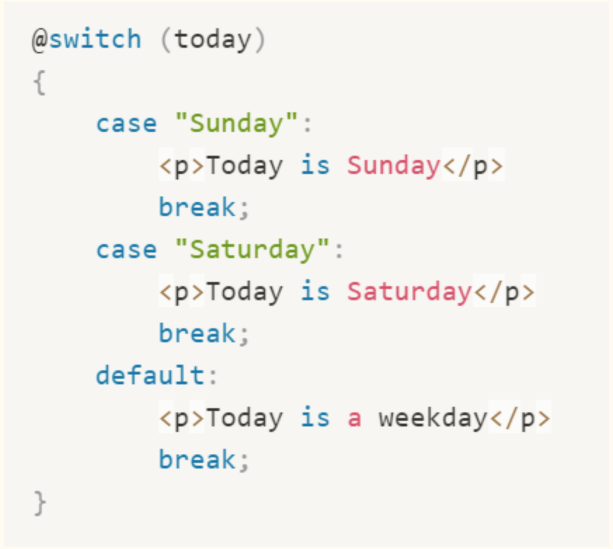
```
@model IEnumerable<Movie>

@foreach(var movie in Model)
{
  <p>@movie.Name</p>
}
```

Flow Control in Razor

- Conditionals: @if, else if, else and @switch



```
@if (value % 2 == 0)
{
    <p>The value is even.</p>
}
else if (value >= 2022)
{
    <p>The value is odd and large.</p>
}
else
{
    <p>The value is odd and small.</p>
}
```



```
@switch (today)
{
    case "Sunday":
        <p>Today is Sunday</p>
        break;
    case "Saturday":
        <p>Today is Saturday</p>
        break;
    default:
        <p>Today is a weekday</p>
        break;
}
```

Flow Control in Razor

- Looping: @for, @foreach, @while, and @do while

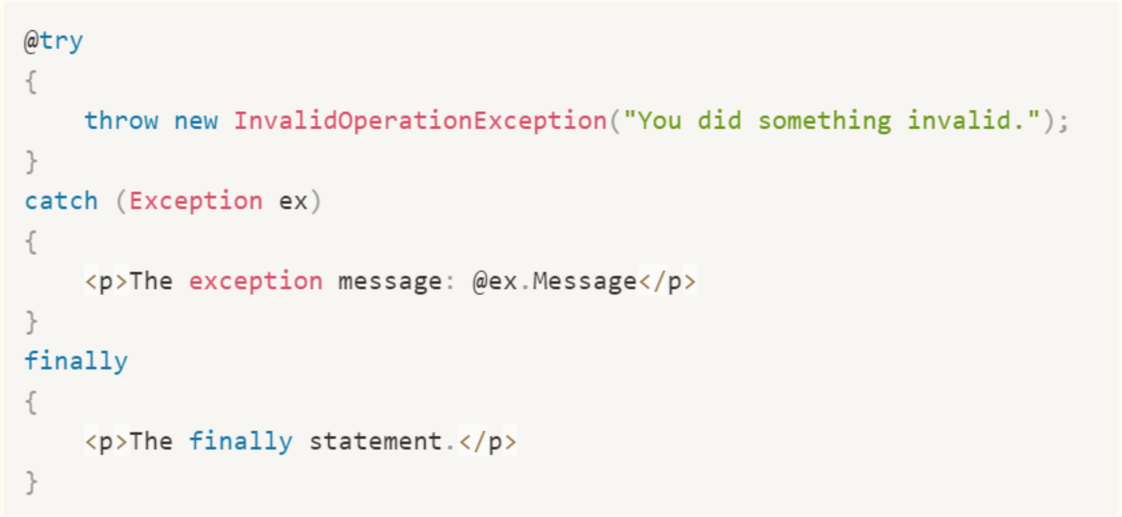
```
@{
    var people = new Person[]
    {
          new Person("Jill", 30),
          new Person("Johnathon", 25),
          new Person("Jason", 25),
    };
}

@for (var i = 0; i < people.Length; i++)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

```
@foreach (var person in people)
{
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
}
```

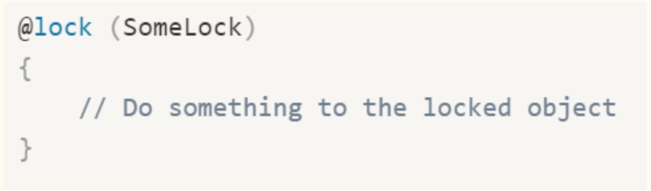
```
@{ var i = 0; }
@while (i < people.Length)
{
    var person = people[i];
    <p>Name: @person.Name</p>
    <p>Age: @person.Age</p>
    i++;
}
```

Other things we can do in Razor

- Exception handling
- Synchronization



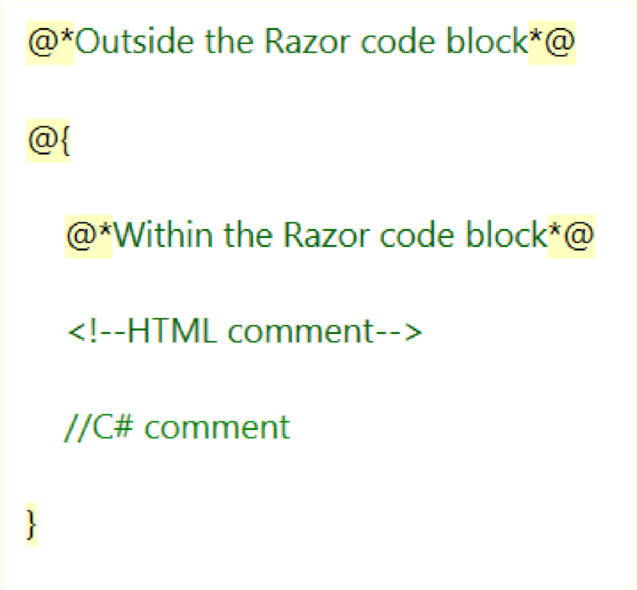
```
@try
{
    throw new InvalidOperationException("You did something invalid.");
}
catch (Exception ex)
{
    <p>The exception message: @ex.Message</p>
}
finally
{
    <p>The finally statement.</p>
}
```



```
@lock (SomeLock)
{
    // Do something to the locked object
}
```

Comments

- Razor comments are removed by the server before the webpage is rendered.
- Razor uses @*your comment*@ to delimit comments.
- In a Razor code block, HTML and C# comments are also supported

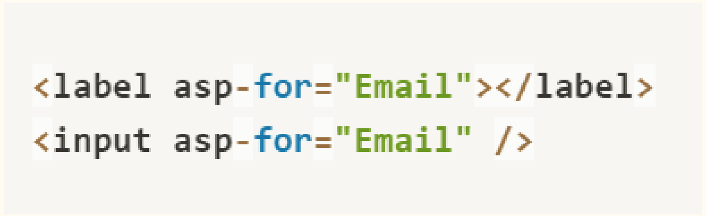


```
@*Outside the Razor code block*@

@{

    @*Within the Razor code block*@

    <!--HTML comment-->

    //C# comment

}
```

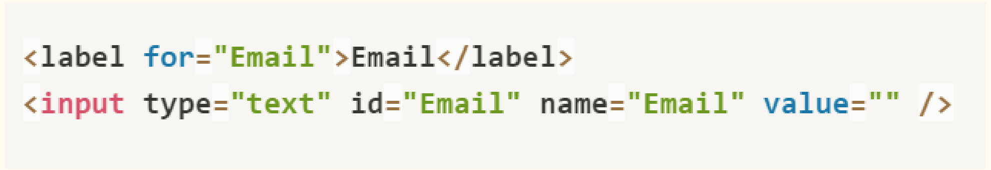
Let’s take a look at the example

Tag Helper

- In ASP.NET Core, Tag Helpers are a way to create custom HTML tags that can be used within Razor views.
- They allow developers to write HTML directly in Razor views while providing a way to include server-side logic that can modify the HTML before it is rendered.
- The Tag Helpers start with an asp- prefix



```
<label asp-for="Email"></label>
<input asp-for="Email" />
```



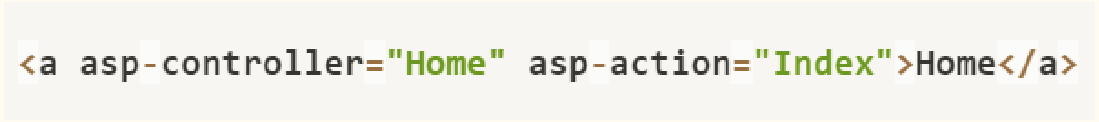
```
<label for="Email">Email</label>
<input type="text" id="Email" name="Email" value="" />
```

Tag Helper

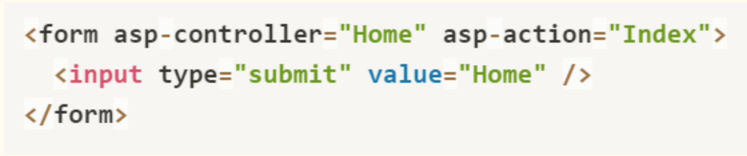
- *asp-controller* and *asp-action* attributes are used in conjunction with the **asp-route** attribute to generate URLs in Razor views.

- *asp-controller* specifies the controller's name that handles the request when the link is clicked.

- *asp-action* specifies the name of the action method that is called on the controller when the link is clicked.



```
<a asp-controller="Home" asp-action="Index">Home</a>
```



```
<form asp-controller="Home" asp-action="Index">
  <input type="submit" value="Home" />
</form>
```

Razor Pages

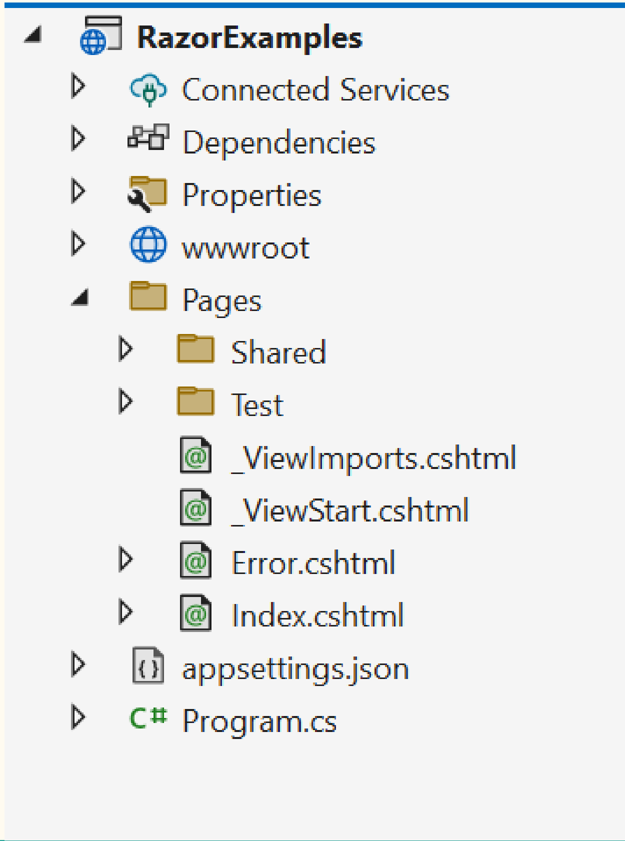
Razor Pages

ASP.NET Razor Pages is a server-side, page-focused framework that enables building dynamic, data-driven websites with clean separation of concerns.

The Razor Pages framework is lightweight and very flexible. It provides the developer with full control over rendered HTML. Razor Pages is the recommended framework for cross-platform server-side HTML generation.

Razor Pages

- wwwroot: contains static assets, like HTML files, JavaScript files, and CSS files.
- Pages: contains Razor page and supporting files, each Razor page is a pair of files:
  - A .cshtml file that has HTML markup with C# code using Razor syntax.
  - A .cshtml.cs file that has C# code that handles page events.
- appsetting.json: a json file that contains configuration data, like connection strings.
- Program.cs: the starter file.



```
RazorExamples
  Connected Services
  Dependencies
  Properties
  wwwroot
  Pages
    Shared
    Test
    _ViewImports.cshtml
    _ViewStart.cshtml
    Error.cshtml
    Index.cshtml
  appsettings.json
  Program.cs
```

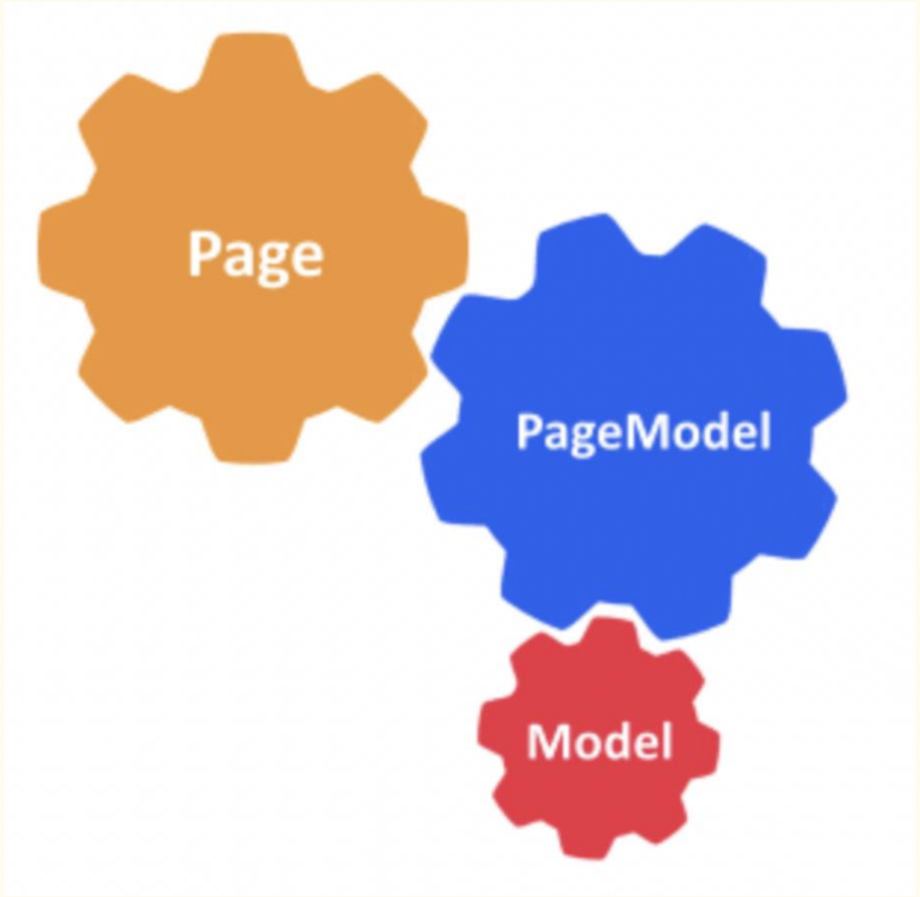
Razor Pages VS MVC

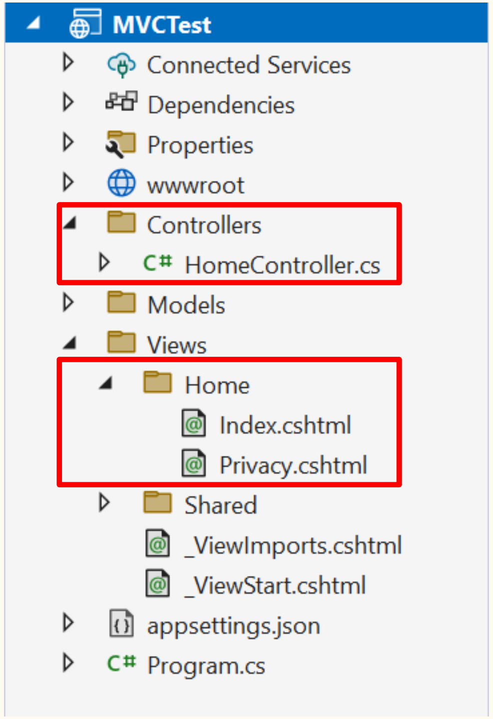
MVC in ASP.NET recap

A typical strongly typed view-based MVC app will use a controller to contain one or more actions. The controller will interact with the domain or data model, and create an instance of a ViewModel class. Then this ViewModel class is passed to the view associated with that action. Using this approach, coupled with the default folder structure of MVC apps, to add a new page to an app requires modifying a controller in one folder.

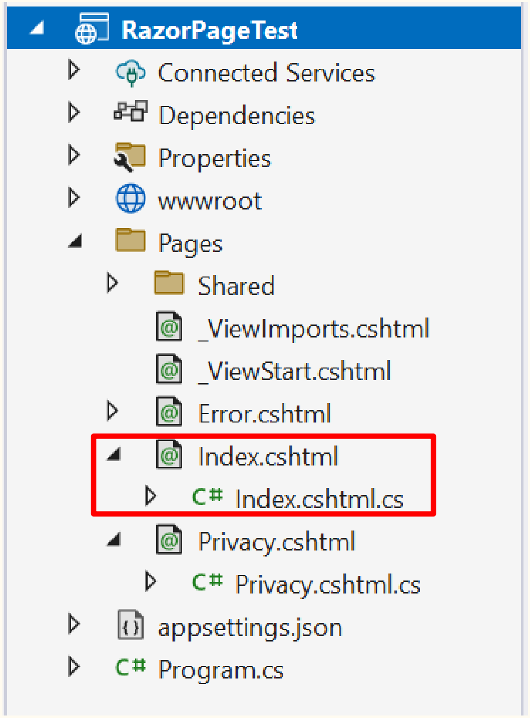
Razor Pages

Razor Pages group together the action (now a handler) and the viewmodel (called a PageModel) in one class, and link this class to the view (called a Razor Page). All Razor Pages go into a Pages folder in the root of the ASP.NET Core project. Razor Pages use a routing convention based on their name and location within this folder. Handlers behave exactly like action methods but have the HTTP verb they handle in their name (for example, OnGet). They also don't necessarily need to return, since by default they're assumed to return the page they're associated with. This tends to keep Razor Pages and their handlers smaller and more focused while at the same time making it easier to find and work with all of the files needed to add or modify a particular part of an app.


Page
PageModel
Model



MVCTest
- Connected Services
- Dependencies
- Properties
- wwwroot
- Controllers
  - HomeController.cs
- Models
- Views
  - Home
    - Index.cshtml
    - Privacy.cshtml
  - Shared
  - _ViewImports.cshtml
  - _ViewStart.cshtml
- appsettings.json
- Program.cs



RazorPageTest
- Connected Services
- Dependencies
- Properties
- wwwroot
- Pages
  - Shared
  - _ViewImports.cshtml
  - _ViewStart.cshtml
  - Error.cshtml
  - Index.cshtml
    - Index.cshtml.cs
  - Privacy.cshtml
    - Privacy.cshtml.cs
- appsettings.json
- Program.cs

Razor Pages vs MVC

- The basic difference between Razor pages and MVC is that the model and controller are added within the Razor Page itself, you don’t need to add code separately.

- If your ASP.NET MVC app makes heavy use of views, you may want to consider migrating from actions and views to Razor Pages.

- However, MVC works well with apps that handle complex logic and have RESTful APIs, we will keep using MVC architecture for future courses.

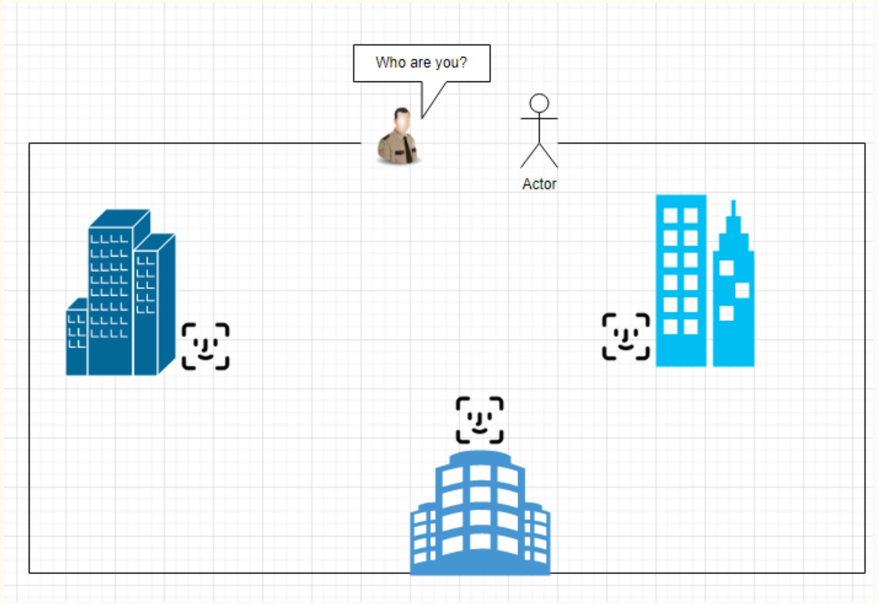
Cookie Authentication

Outline

- Authentication and Authorization
- Session/Cookie
- Cookie Authentication
- Filter

Authentication & Authorization

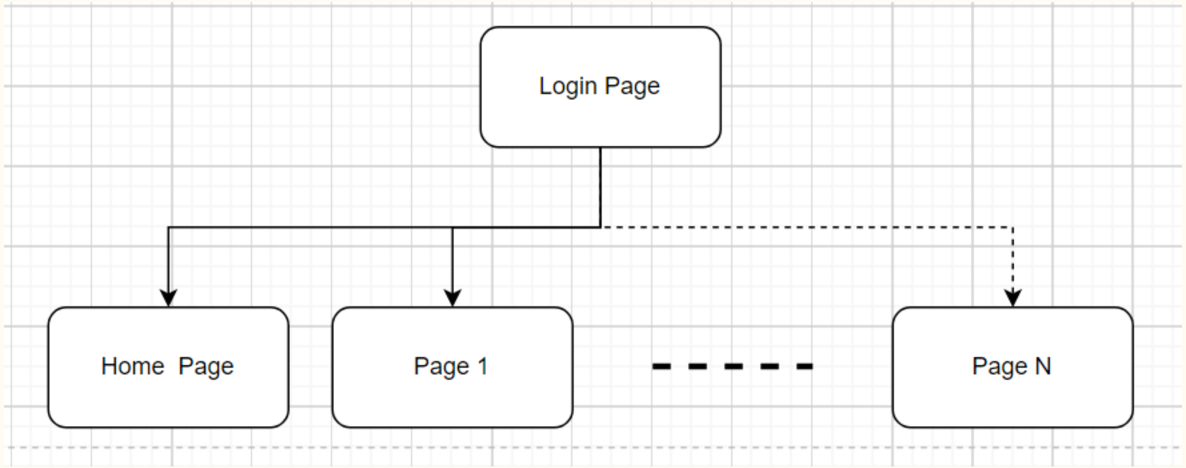
Q: How can I enter the confidential campus?



Authentication & Authorization

- **Authentication** verifies the identity of a user or service
- **Authorization** determines their access rights
- **Security Context** contains all your identity information that is relevant to the facility

Authentication & Authorization



HTTP is Stateless!

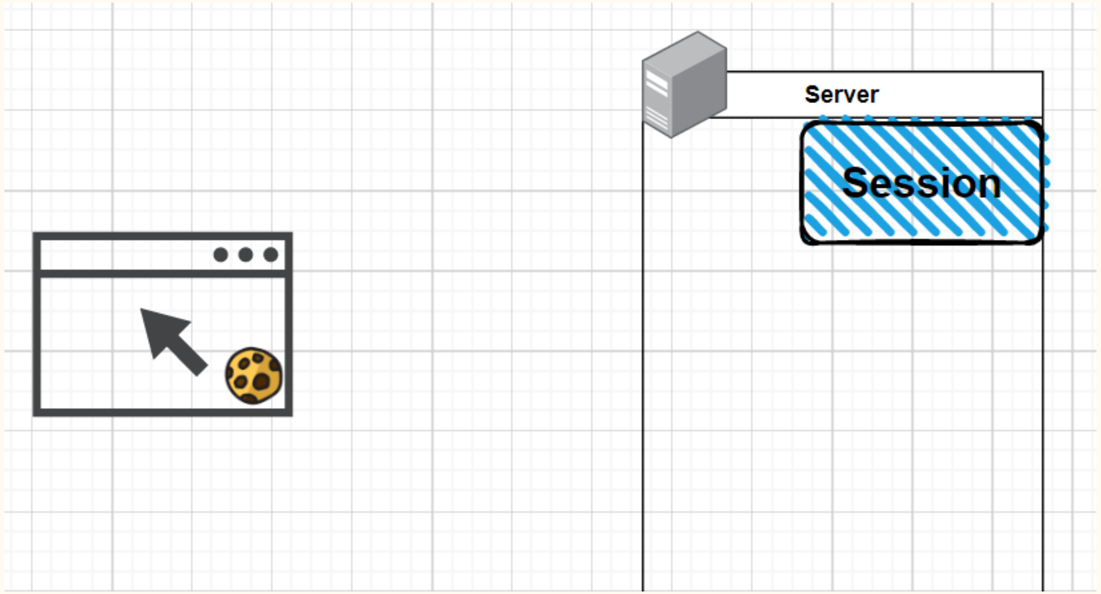
The problem with HTTP

- HTTP is ***stateless*** — we don’t know if two HTTP requests come from the same user or not.

In order to associate a request with any other request, we need a way to store user data between HTTP requests

- **URL parameters** — Passing the same credential with all request**(Unsafe)**
- **Cookies** — Storing the credential on the client side **(Unsafe)**
- **Session** — Storing the credential on the server side, give it an "id", and let the client only know that id (and pass it back at every HTTP request). That is SESSION!

Session & Cookie



Cookie

- Cookie is a small piece of information that is sent by the web server in the **response header** and gets stored in the browser cookies
- It is a **key value pair** sent by the server to the client
- Cookies store data across requests.
- This pair is automatically attached with every request to the server, so their size should be kept to a minimum
- Cookies are often used for personalization, where content is customized for a known user

Session

- Session state is an ASP.NET Core scenario for the **storage of user data while the user browses a web app**
- Session is accessed via **HttpContext.Session**
- Session cannot be shared across browsers

Cookie Authentication

If the user information is stored in the cookie, it is not safe, because this information will be stored on the client side (on the browser).
For security reasons, the user information is saved in the session on the server side, but the ***session id*** will be stored in the cookie so that when the cookie is passed between the browser and the server, the server can find the corresponding session through the ***session id*** in the cookie, so that the user does not have to log in again and the website can recognize the user.

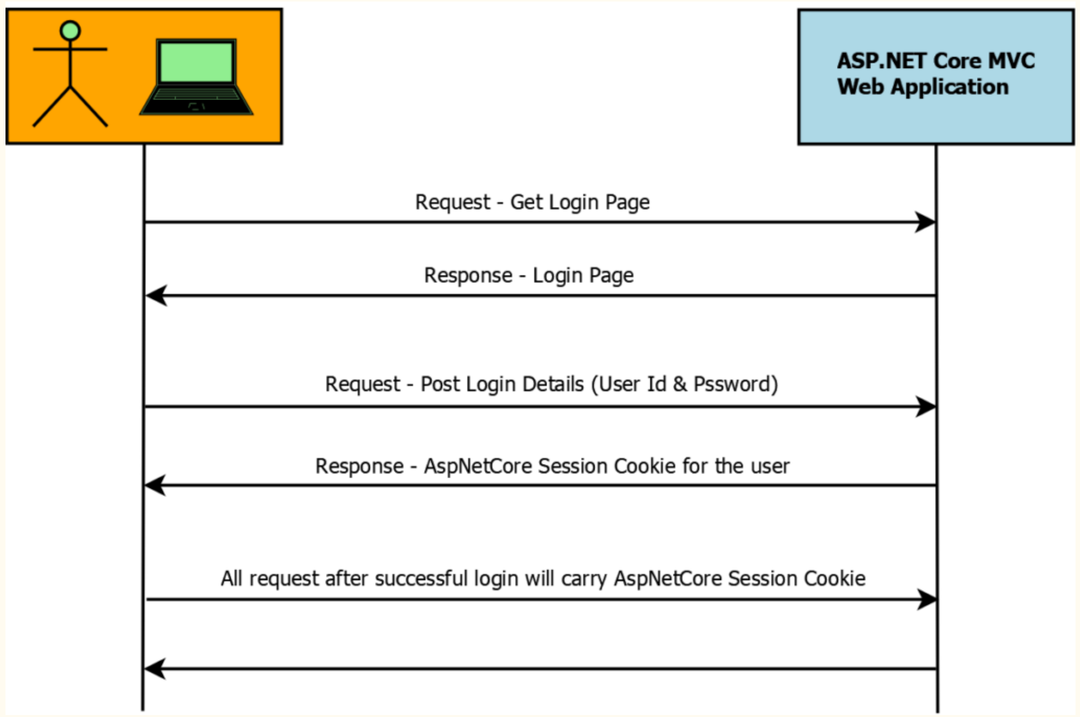
- Session cookies are deleted when the browser session ends.
- If a cookie is received for an expired session, a new session is created that uses the same session cookie

Cookie Authentication

Cookie authentication in ASP.NET Core web application is the popular choice for developers to implement authentication in most customer-facing web applications and is also easy to implement in ASP.NET Core as it is provided out of the box without the need to reference any additional NuGet packages.

ASP.NET Core provides a **cookie authentication mechanism** which on login serializes the user details in form of **claims** into an encrypted cookie and then sends this cookie back to the server on subsequent requests which gets validated to recreate the user object from claims and sets this user object in the HttpContext so that it is available & is valid only for that request.

Cookie Authentication

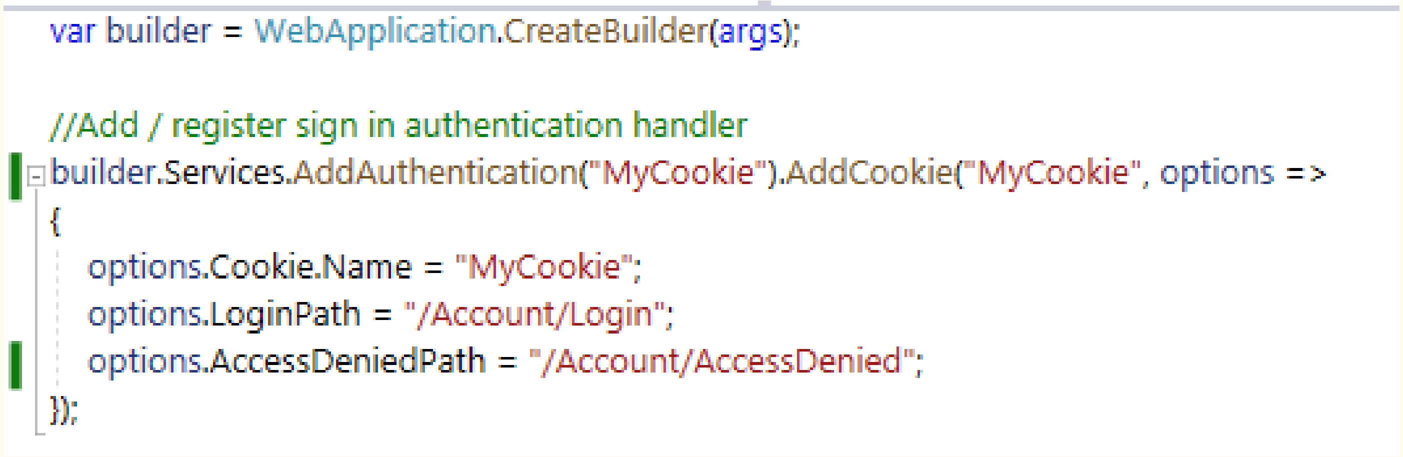

ASP.NET Core MVC Web Application
Request - Get Login Page
Response - Login Page
Request - Post Login Details (User Id & Pssword)
Response - AspNetCore Session Cookie for the user
All request after successful login will carry AspNetCore Session Cookie

Using Cookie Authentication

1. Add authentication middleware with the AddAuthentication and AddCookie methods
2. Specify the app must use authentication and authorization
3. Apply [Authorize] attribute on the controllers and actions that require the cookie authorization

Step 1: Add Authentication Middleware

In your Program.cs file:



```
var builder = WebApplication.CreateBuilder(args);

//Add / register sign in authentication handler
builder.Services.AddAuthentication("MyCookie").AddCookie("MyCookie", options =>
{
    options.Cookie.Name = "MyCookie";
    options.LoginPath = "/Account/Login";
    options.AccessDeniedPath = "/Account/AccessDenied";
});
```

On .NET 5.0 or the previous version, we have to do this configuration inside the **Startup.cs** and inside its **ConfigureServices()** method

Cookie Authentication Options

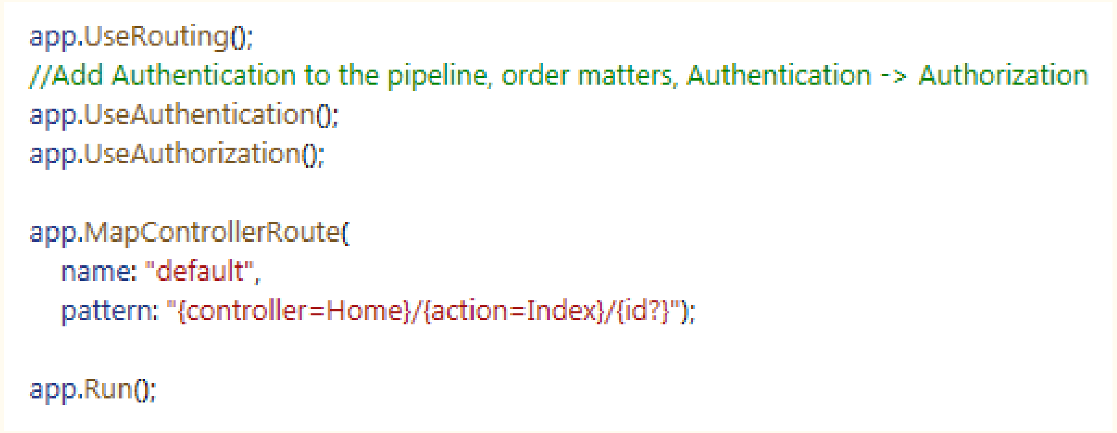
- **Cookie.Name** – Name of the Cookie
- **LoginPath** – is used by the handler for the redirection target when handling ChallengeAsync
- **SlidingExpiration** – is set to true to instruct the handler to re-issue a new cookie with a new expiration time any time it processes a request
- **AccessDeniedPath** – is used by the handler for the redirection target when handling ForbidAsync

Authentication Properties Options

- **IsPersistent** – Gets or sets whether the authentication session is persisted across multiple requests.
- **ExpiresUtc** – Gets or sets the time at which the authentication ticket expires.
- **AllowRefresh** – Gets or sets if refreshing the authentication session should be allowed.
- **IssuedUtc** – Gets or sets the time at which the authentication ticket was issued.

Step 2: UseAuthentication() & UseAuthorization()

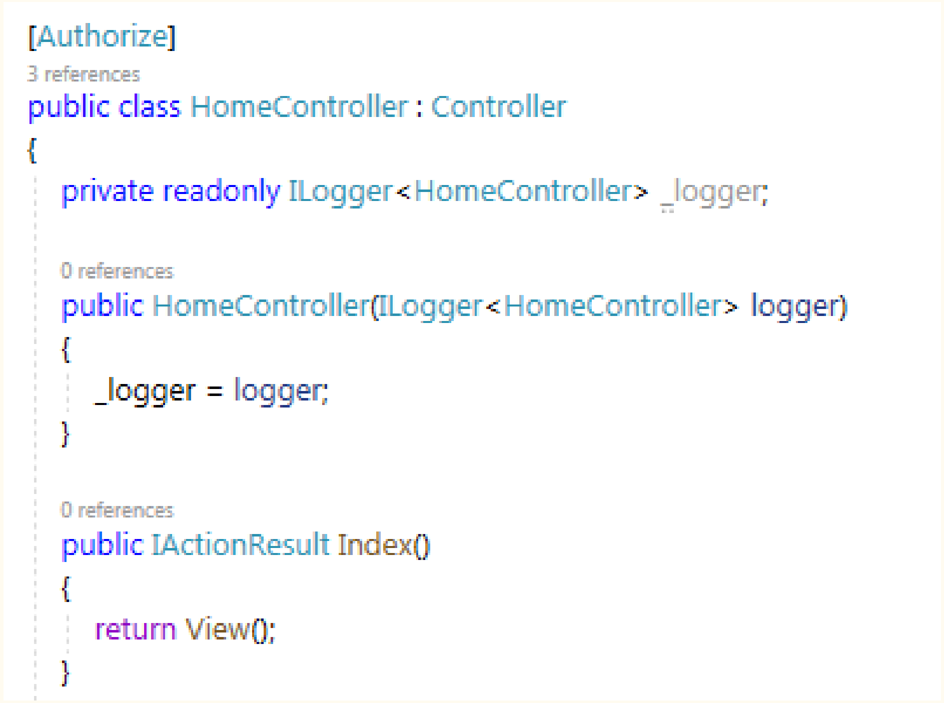
In your Program.cs file:



```
app.UseRouting();
//Add Authentication to the pipeline, order matters, Authentication -> Authorization
app.UseAuthentication();
app.UseAuthorization();

app.MapControllerRoute(
    name: "default",
    pattern: "{controller=Home}/{action=Index}/{id?}");

app.Run();
```

On .NET 5.0 or earlier versions, those will be included within Configure() method of Startup.cs. This tells the App to use the authentication & authorization

Step 3: Apply [Authorize] attribute



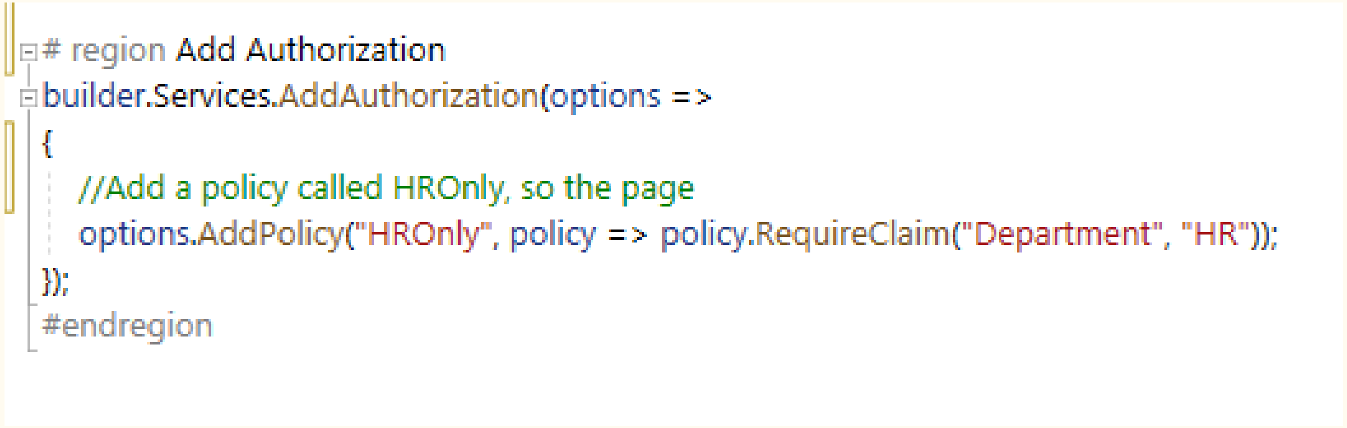
```
[Authorize]
3 references
public class HomeController : Controller
{
    private readonly ILogger<HomeController> _logger;

    0 references
    public HomeController(ILogger<HomeController> logger)
    {
        _logger = logger;
    }

    0 references
    public IActionResult Index()
    {
        return View();
    }
```

[Authorize] Attribute

- When we place the Authorize attribute on the controller itself, the authorize attribute applies to all of the actions inside.
- The MVC framework will not allow a request to reach an action protected by this attribute unless the user passes an authorization check.
- By default, if you use no other parameters, the only check the Authorize attribute will make is a check to ensure the user is logged in, so we know their identity.
- But you can use parameters to specify any fancy custom authorization policy that you like.

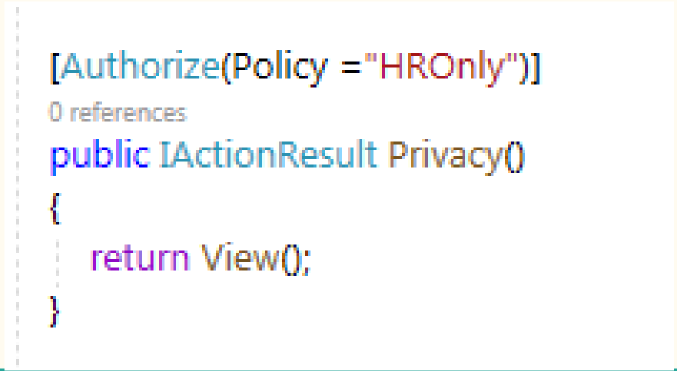
Add more Policy

- Program.cs

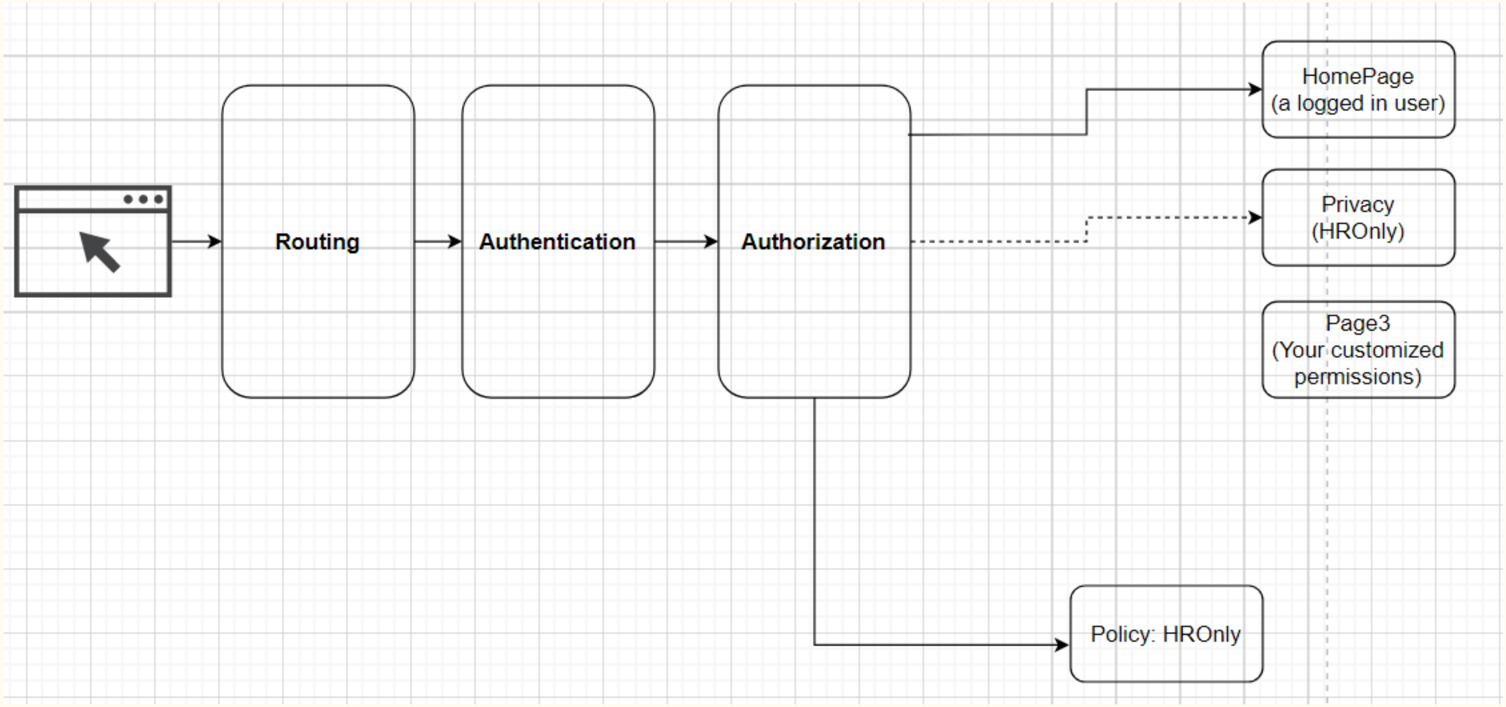


```
# region Add Authorization
builder.Services.AddAuthorization(options =>
{
    //Add a policy called HROnly, so the page
    options.AddPolicy("HROnly", policy => policy.RequireClaim("Department", "HR"));
});
#endregion
```

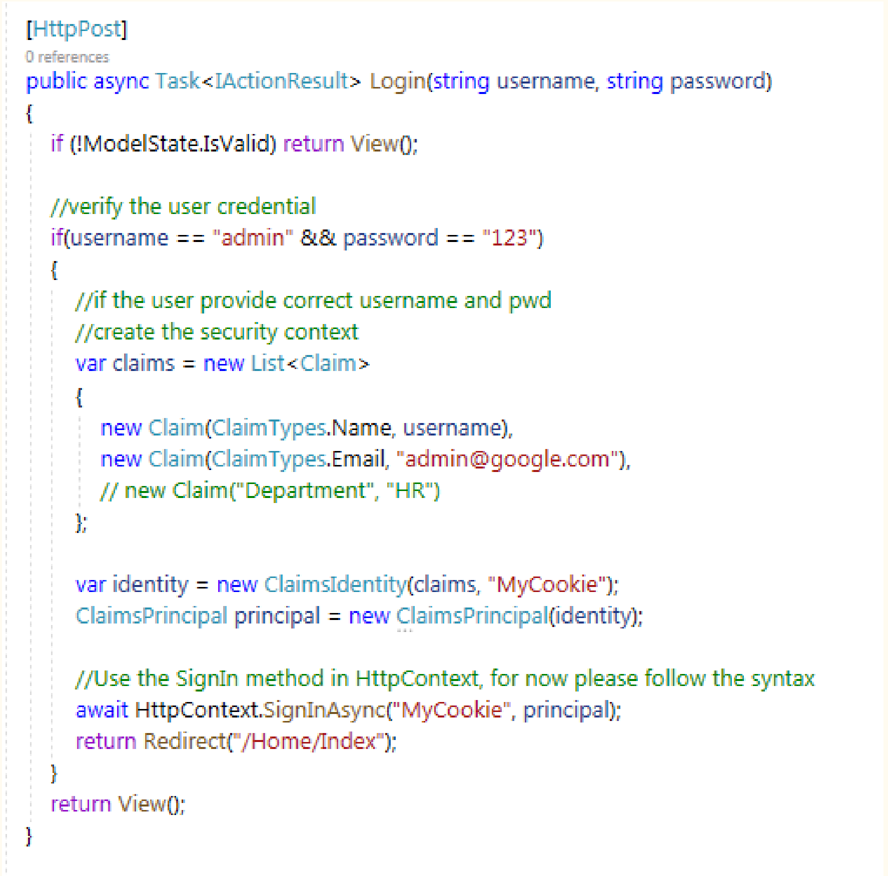
- Controller.cs



```
[Authorize(Policy ="HROnly")]
0 references
public IActionResult Privacy()
{
    return View();
}
```


Routing
Authentication
Authorization
HomePage
(a logged in user)
Privacy
(HROnly)
Page3
(Your customized
permissions)
Policy: HROnly

Login

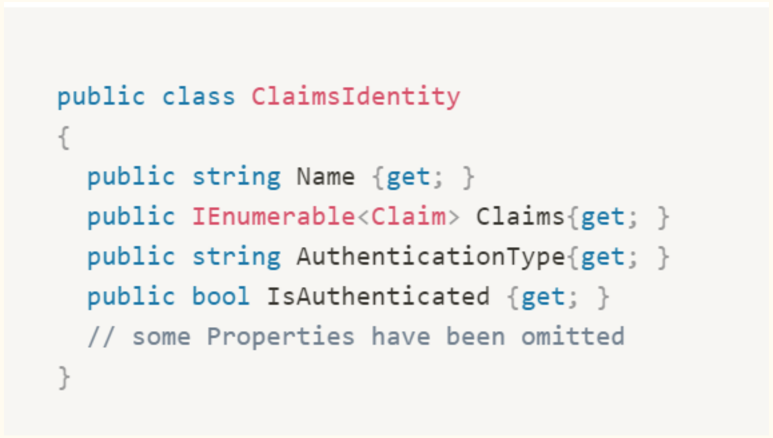

```
[HttpPost]
0 references
public async Task<IActionResult> Login(string username, string password)
{
    if (!ModelState.IsValid) return View();

    //verify the user credential
    if(username == "admin" && password == "123")
    {
        //if the user provide correct username and pwd
        //create the security context
        var claims = new List<Claim>
        {
            new Claim(ClaimTypes.Name, username),
            new Claim(ClaimTypes.Email, "admin@google.com"),
            // new Claim("Department", "HR")
        };

        var identity = new ClaimsIdentity(claims, "MyCookie");
        ClaimsPrincipal principal = new ClaimsPrincipal(identity);

        //Use the SignIn method in HttpContext, for now please follow the syntax
        await HttpContext.SignInAsync("MyCookie", principal);
        return Redirect("/Home/Index");
    }
    return View();
}
```

Claims

- **Claims** are the foundation behind claims-based authentication (who would have guessed). A claim is simply **a piece of information about a subject**. A claim does not dictate what a subject can, or cannot do.
- The term “subject” is used because claims are not restricted to only describing users. Claims can be about an application, service, or device.
- Some examples of claims a subject may have are:
  - Username
  - Email
  - IP Address
  - Location

ClaimsIdentity

Claims **representing the same subject** can be grouped together and placed in a **ClaimsIdentity**.

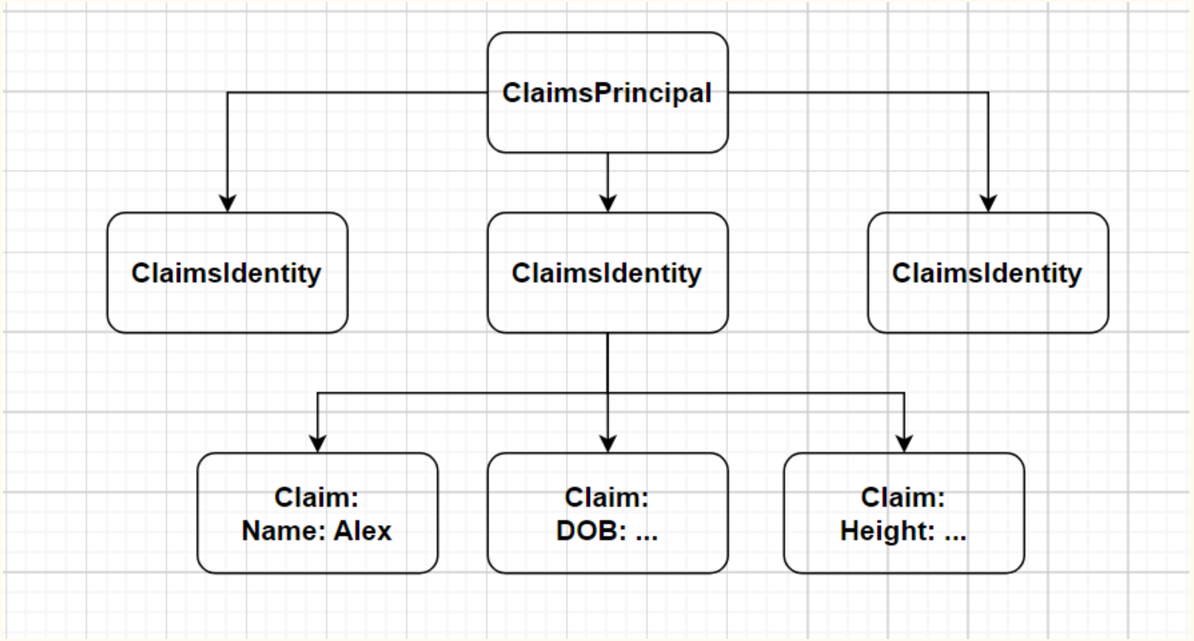


```
public class ClaimsIdentity
{
  public string Name {get; }
  public IEnumerable<Claim> Claims{get; }
  public string AuthenticationType{get; }
  public bool IsAuthenticated {get; }
  // some Properties have been omitted
}
```

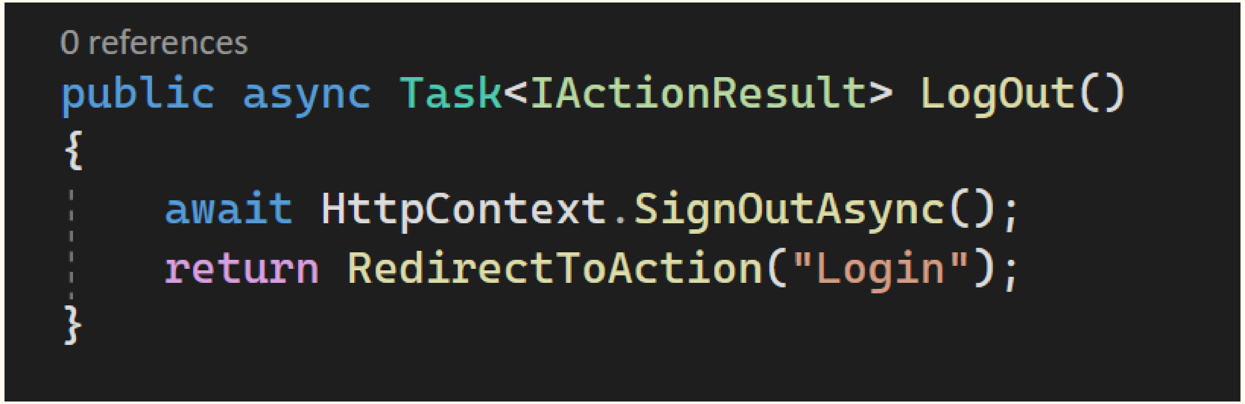
ClaimsPrincipal

A principal object **represents the security context of the user** on whose behalf the code is running, including that user's identity (IIdentity) and any roles to which they belong.

By using a **ClaimsPrincipal** we can group the user identity, and device identity into one context without having to duplicate any info.

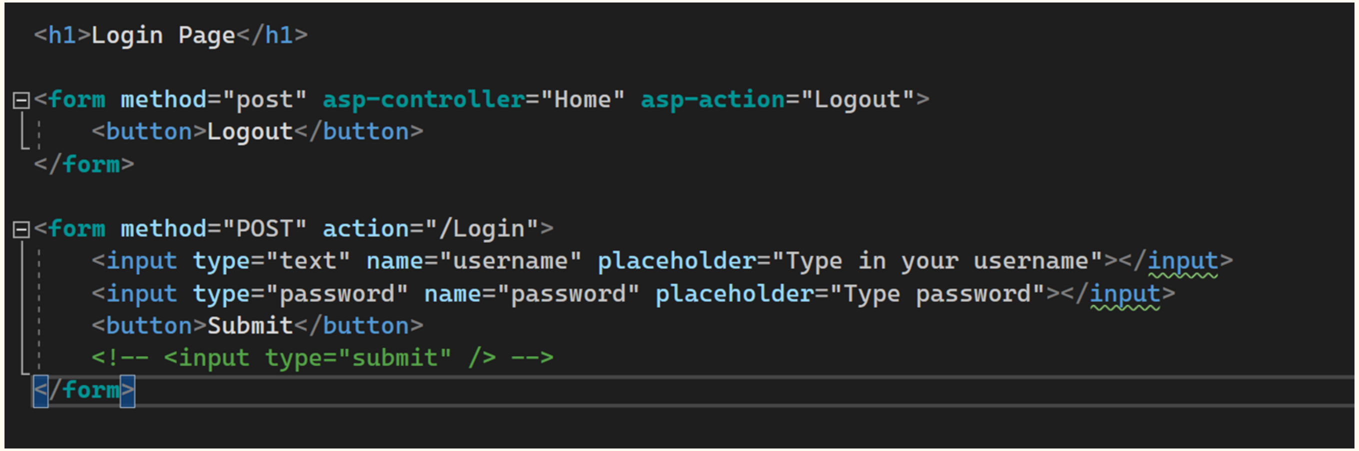

ClaimsPrincipal
ClaimsIdentity
ClaimsIdentity
ClaimsIdentity
Claim:
Name: Alex
Claim:
DOB: ...
Claim:
Height: ...

Logout



```
0 references
public async Task<IActionResult> LogOut()
{
    await HttpContext.SignOutAsync();
    return RedirectToAction("Login");
}
```

Login/Logout View



```
<h1>Login Page</h1>

<form method="post" asp-controller="Home" asp-action="Logout">
    <button>Logout</button>
</form>

<form method="POST" action="/Login">
    <input type="text" name="username" placeholder="Type in your username"></input>
    <input type="password" name="password" placeholder="Type password"></input>
    <button>Submit</button>
    <!-- <input type="submit" /> -->
</form>
```

Filter

In MVC, controllers have many action methods and these action methods generally have a one-to-one relationship with UI controls such as clicking a button or a link.

But many times we would like to perform some action before or after a particular operation. For achieving this functionality, ASP.NET MVC provides feature to add **pre and post action behaviors on controller's action methods.**

Filter Types

ASP.NET/ASP.NET Core MVC framework supports the following filter types:

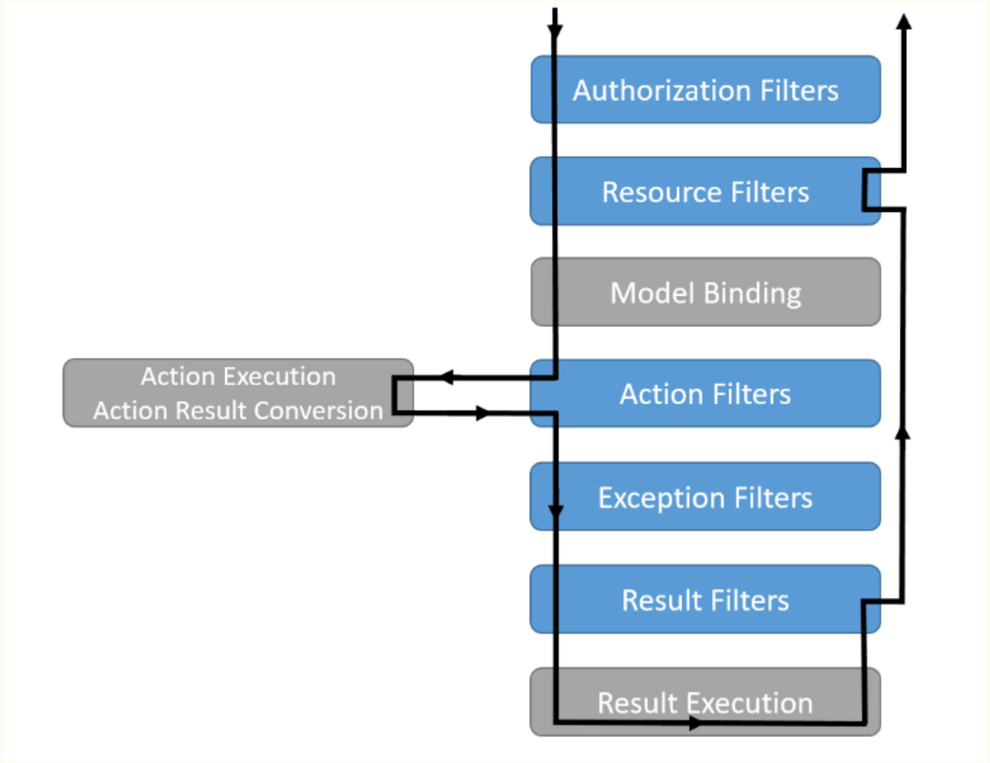
- **Authorization filters:**
  - Are the **first filters** run in the filter pipeline.
  - Control access to action methods.
  - Have a before method, but no after method.
- **Resource filters:**
  - Implement either the *IResourceFilter* or *IAsyncResourceFilter* interface.
  - Execution wraps most of the filter pipeline.
  - Only Authorization filters run before resource filters.
- **Action filters:**
  - Implement either the *IActionFilter* or *IAsyncActionFilter* interface.
  - Their execution surrounds the execution of action methods.

Filter Types

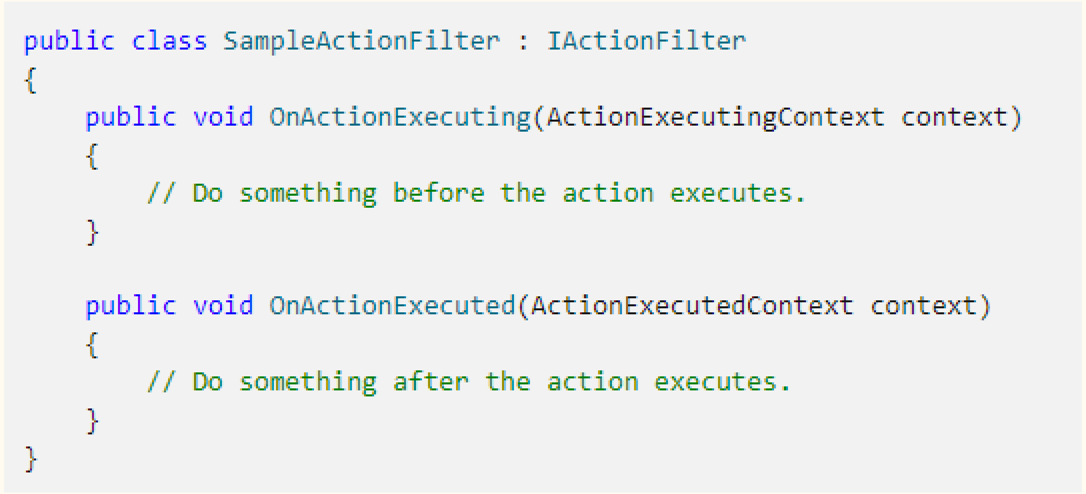
ASP.NET/ASP.NET Core MVC framework supports the following filter types:

- **Exception filters** apply global policies to unhandled exceptions that occur before the response body has been written to.
- **Result filters**:
  - Run immediately before and after the execution of action results.
  - Run only when the action method executes successfully.
  - Are useful for logic that must surround view or formatter execution.

Filter Order



Filter Implementation (Action Filter)



```
public class SampleActionFilter : IActionFilter
{
    public void OnActionExecuting(ActionExecutingContext context)
    {
        // Do something before the action executes.
    }

    public void OnActionExecuted(ActionExecutedContext context)
    {
        // Do something after the action executes.
    }
}
```

Any Questions?